Gem #51: Safe and Secure Software: Chapter 11, Certified Safe with SPARK

Author: John Barnes

Abstract: Gem #51 is the eleventh chapter of John Barnes' new booklet: Safe and Secure Software: An Introduction to Ada 2005.

Let's get started…

For some applications, especially those that are safety-critical or security-critical, it is essential that the program be correct, and that correctness be established rigorously through some formal procedure. For the most severe safety-critical applications the consequence of an error can be loss of life or damage to the environment. Similarly, for the most severe security-critical applications the consequence of an error may be equally catastrophic such as loss of national security, commercial reputation or just plain theft.

Applications are graded into different levels according to the risk. For avionics applications the DO-178B standard [1] defines the following:

level E none: no problem; e.g. entertainment system fails? – could be a benefit!

level D minor: some inconvenience; e.g. automatic lavatory system fails.

level C major: some injuries; e.g. bumpy landing, cuts and bruises.

level B hazardous; some dead; e.g. nasty landing with fire.

level A catastrophic; aircraft crashes, all dead; e.g. control system fails.

As an aside, note that although a failure of the entertainment system in general is level E, if the failure is such that the pilot is unable to switch it off (perhaps in order to announce something unpleasant) then that failure is at level D.

For the most demanding applications, which require certification by an appropriate authority, it is not enough for a program to be correct. The program also has to be shown to be correct and that is much more difficult.

This chapter gives a very brief introduction to SPARK. This is a language based on a subset of Ada which was specifically designed for the writing of high integrity systems. Although technically just a subset of Ada with additional information provided through Ada comments, it is helpful to consider SPARK as a language in its own right which, for convenience, uses a standard Ada compiler, but which is amenable to a more formal treatment than the full Ada language. Analysis of a SPARK program is carried out by a suite of tools of which the most important are the Examiner, Simplifier, and Proof Checker.

Chapter 11 is available in full (in PDF) at the Ada Gems page at AdaCore.

Related Source Code

Gem #52: Scripting Capabilities in GNAT (Part 1)

Author: Emmanuel Briot, AdaCore

Abstract: Ada Gem #52 — Programming languages are, at least in the mind of most programmers, divided into two categories: scripting languages and others. The dividing line is not always obvious, and generally has to do with the size of the application, whether the language is interpreted or compiled, and whether doing operations such as manipulating external processes is easy.

It's safe to say, though, that Ada is almost never classified as a scripting language. This Gem demonstrates a few of the packages that are part of the GNAT distribution and that provide helpful scripting features to accomplish tasks that would often be thought to be in the domain of languages specialized for scripting. Portability is of course a major advantage of choosing Ada!

Let's get started…

The first thing a program generally has to do is to parse its command line to find out what features a user wants to activate. Standard Ada provides the Ada.Command_Line package, which basically gives you access to each of the arguments on the command line. But GNAT provides a much more advanced package, GNAT.Command_Line, which helps to manipulate those arguments, so that you can easily extract switches, their (possibly optional) arguments and any remaining arguments. Here is a short example of using that package:

```
with GNAT.Command_Line;  use GNAT.Command_Line;
procedure Main is
begin
  loop
     case Getopt ("h f: d? e g") is
       when 'h' => Display_Help;
       when 'f' => Set_File (Parameter);
       when 'd' => Set_Directory (Parameter);
       when 'e' | 'g' => Do_Something (Full_Switch);
       when others => exit;
     end case;
  end loop;
exception
  when Invalid_Switch | Invalid_Parameter =>
     Display_Help;
end Main;
```

This application accepts several switches. -f requires an extra argument, whereas -d has an optional argument. The application can be called in several ways: for example, "-ffile -e -g", "-f file -e -g", or "-f file -eg". GNAT.Command_Line is extremely flexible in what it accepts from the user, which can help to make the application easier to use.

One other convenient package is GNAT.Regpat, which provides the power of regular expression processing in Ada programs. This package was already discussed in a previous Gem related to searching, so we will not detail it again here (see Gem #25).

It is often the case that scripts need to parse text files. Although the standard package Ada.Text_IO gives access to such files, it has several drawbacks. First, it is rather slow. In fact, the Ada standard forces this package to manage a number of additional items of information internally, which can add considerable overhead. Also, the file is read chunk by chunk, which on most systems is slow (and requires lots of blocking system calls). Where possible, it is better to use more efficient standard packages such as Stream_IO. Furthermore, Ada.Text_IO doesn't provide any parsing capability.

The GNAT Reusable Components (initially released to customers in July 2008) provide the package GNATCOLL.Mmap. This package typically uses more efficient system calls to read the file, and results in much greater speed if your application is reading lots of text files.

```
with GNATCOLL.Mmap;  use GNATCOLL.Mmap;
procedure Main is
   File : Mapped_File := Open_Read ("filename");
   Str  : Str_Access;
begin
   Read (File);  --  read whole file at once
   Str := Data (File);
   --  you are now manipulating an in-memory version of the file
   Close (File);
end Main;
```

This package also provides support for only reading part of a file at once in memory, which is important if you are manipulating huge files. But most often it is more efficient simply to read the whole file at once.

Regarding parsing facilities, another package comes to the programmer's aid: GNAT.AWK. This package provides an interface similar to the standard awk utility on Unix systems. Specifically, one of its modes is an implicit loop over the whole file. Pattern matching is applied to each line in turn, and when the pattern matches, a callback action is taken. This means you no longer have to do all the parsing yourself. Several types of iterators are provided, and this Gem illustrates only one. See the interface of g-awk.ads for more examples.

```
with GNAT.AWK;  use GNAT.AWK;
procedure Main is
   procedure Action1 is
   begin
      Put_Line (Field (2));
   end Action2;

   procedure Default_Action is
```

```
   begin
      null;
   end Default_Action;
begin
   Register (1, "saturn|earth", Action1'Access);
   Register (1, ".*", Default_Action'Access);
   Parse (";", "filename");
end Main;
```

Each line in the file denoted by “filename” contains groups of fields with semicolons used to separate the groups. Actions are registered for different values of the first field in a group. When the first field matches “saturn” or “earth”, the appropriate action is called, and the second field is printed. For all other planets nothing is done. This code is certainly bigger than the equivalent Unix shell command using awk, but is still very reasonably sized. Of course, if you make a mistake in your program, it is very likely that the compiler will help you find it (ever tried to debug a 20-line awk program, not to mention even longer programs, where quotes rapidly become an issue?)

An upcoming Gem will discuss additional scripting features, focusing on manipulation of external processes.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #54: Scripting Capabilities in GNAT (Part 2)

Author: Emmanuel Briot, AdaCore

Abstract: In Gem #52, we saw various facilities that GNAT provides to make Ada a scripting language (sort of). We discussed GNAT.Command_Line, GNATCOLL.Mmap, GNAT.Regpat and GNAT.AWK. This Gem continues the discussion of scripting capabilities, and focuses on interaction with external processes.

Let's get started…

A lot of applications have a need to execute commands on the machine. There are several use cases: a program might wish to wait for a command to complete (for instance the command generates a file that the application then needs to parse); it might spawn a command in the background and continue to execute in the meantime; or it might need to interact with an external application (sending data to its input stream and reading its output).

Spawning a process that the application doesn't need to interact with is easy. The GNAT run-time provides a package GNAT.OS_Lib containing a number of subprograms that are low-level interfaces to the system. In particular, the package provides several subprograms with names Spawn and Non_Blocking_Spawn which, as their name indicates, will spawn an external application, and wait or not for its completion.

```
with GNAT.OS_Lib;  use GNAT.OS_Lib;

procedure Main is
   Command : constant String := "myapp -f 'a string with spaces'";
   --  We assume the slightly more complex case where the arguments of
the
   --  command are part of the same string (this is generally the case
when
   --  the command is provided interactively by the user).

   Args        : Argument_List_Access;
   Exit_Status : Integer;
   Pid         : Process_Id;
   Success     : Boolean;

begin
   --  Prepare the arguments. Splitting properly takes quotes into
account.

   Args := Argument_String_To_List (Command);

   --  Spawn the command and wait for its possible completion

   if Background_Mode then
      Pid := Non_Blocking_Spawn
         (Program_Name => Args (Args'First).all,
          Args         => Args (Args'First + 1 .. Args'Last));
```

```
      --  We could also wait for completion:
      --  Wait_Process (Pid, Success);

   else
      Exit_Status := Spawn
         (Program_Name => Args (Args'First).all,
          Args         => Args (Args'First + 1 .. Args'Last));
   end if;

   --  Free memory

   Free (Args);

end Main;
```

While discussing GNAT.OS_Lib, it is worth investigating its remaining subprograms. They provide system-independent file manipulation (checking whether a file exists, whether it is a directory or a symbolic link, access to the date of the last modification for the file, copying, deleting, and renaming files, etc.). One very interesting subprogram is Normalize_Pathname, which is able to resolve symbolic links and convert the casing of file names depending on whether they can be accessed using a unique name or with case insensitivity. This provides a convenient way to check whether two file names are the same (handling "..", symbolic links, and case resolution is quite complicated to get right and system-independent, and GNAT.OS_Lib takes care of it for you).

Among the three use cases we mentioned at the beginning, the most complicated one is where the application needs to interact with a process. Such interaction is provided through the GNAT.Expect package. Here, you can spawn a process, and, while it is running, send it input and read its output. Communication between the two processes is done through pipes. Another implementation, using an approach based on ttys (pseudo terminals), is planned to be made available as part of the GNAT Reusable Components, and is designed to provide a closer emulation of what happens when you spawn a process from a terminal.

Here is an example of using GNAT.Expect:

```
with GNAT.Expect;  use GNAT.Expect;

procedure Main is
   Command : constant String := "gdb myapp"; --  Let's spawn a debugger
session.
   Pd      : Process_Descriptor;
   Args    : Argument_List_Access;
   Result  : Expect_Match;

begin
   --  First we spawn the process. Splitting the program name and the
   --  arguments is done as in the previous example, using
   --  Argument_String_To_List. Remember to free the memory at the end.

   Args := Argument_String_To_List (Command);
```

```
   Non_Blocking_Spawn
      (Pd,
       Command     => Args (Args'First).all,
       Args        => Args (Args'First + 1 .. Args'Last),
       Buffer_Size => 0);

   --  The debugger is now running. Let's send a command.

   Send (Pd, "break 10");

   --  Then let's read the output of the debugger.
   --  Here, we are expecting any possible non-empty output (hence the
   --  ".+" regexp). We might in fact be expecting a file name that
   --  gdb uses to confirm a breakpoint. The regexp would be something
   --  like:
   --   "file (.*), line (\d+)"

   Expect (Pd, Result, Regexp => ".+", Timeout => 1_000);

   case Result is
      when Expect_Timeout => …;  --  gdb never replied
      when 1  => …;              --  the regexp matched
          --  We then have access to all the output of gdb since the
          --  last call to Expect, through the Expect_Out subprogram.
   end case;

   --  When we are done, terminate gdb:

   Close (Pd);
end Main;
```

This example really only briefly touched on the capabilities of GNAT.Expect. As a side note, the whole of GPS, our integrated development environment uses this package to interact with external processes.

Related Source Code

Gem #55: Introduction to Ada / Java Interfacing

Author: Quentin Ochem, AdaCore

Abstract: Interfacing Ada and Java is a very tricky problem. As opposed to C, C++, or Fortran, the two languages execute on two different environments, Java on the JVM and Ada directly on the underlying OS. For this reason, it is not possible to directly bind Java functions to natively compiled Ada through a pragma Import. Two solutions are offered to the developer: either compiling the code directly to Java bytecode, using GNAT for the JVM, or using the Java Native Interface (JNI), allowing communication between the native and JVM environments. In this Gem we will take a look at the latter.

Using this JNI layer manually is very error-prone and tedious. Fortunately, AdaCore provides a set of tools for automating the interface generation, through GNAT-AJIS. This Gem is the first of a series showing how this toolset can be used to create a mixed Ada / Java application.

Let's get started…

Suppose we have an API (Application Program Interface) on the Ada side that we would like to call from Java. This API may be based on a set of types and functions, for example:

```
package API is

   type R is record
      F1, F2 : Integer;
   end record;

   procedure Print (Str : String; V : R);

end API;
```

Our purpose is to create an object of type R from Java, and then pass that object to the Ada Print procedure, along with a string parameter.

Generating the binding code

Now we need to generate the Java code as well as the extra JNI layer that will connect to the Ada code. This can be done using the ada2java tool, called as follows:

```
ada2java api.ads –b test –o ada –c java –L my_lib
```

Since the tool only generates the binding to the package, it only needs access to the specification (the api.ads file).

Then, for organizational purposes, we provide the name of the base package for all the generated Java classes, using the –b flag. Here, all the generated classes and packages will be children of the Java package “test”.

The –o and –c flags define the directory where, respectively, the Ada and the Java generated code will be stored. Here, we put the Ada code in the directory ada/ and the java code in the directory java/.

Finally, the main Ada code (here, the package My_Package) and the generated glue code needs to be compiled in a shared library that will be loaded by Java. In order to simplify the process, it is possible to generate a GNAT project file for input to gprbuild that will handle compilation of this library, by using the –L switch.

Writing the Java code

If you take a look at the content of the java/ directory, you willl find a bunch of classes, in particular:

```
test.API.R
test.API.API_Package
test.Standard.AdaString
```

These are all the classes needed by our application. test.API.R maps to the Ada R object, test.API.API_Package contains all the subprogram declarations that come from the API package and cannot be defined as members of other types, and test.Standard.AdaString is an automatically generated class that maps the standard String type.

Observe that the test.API.R class comes with accessors and modifiers for the F1 and F2 fields, and includes a default constructor. Note also that test.Standard.AdaString comes with a constructor based on java.lang.String.

It is now time to write the main Java class:

```
import test.API.R;
import test.API.API_Package;
import test.Standard.AdaString;

public class My_Main {

public static void main (String [] argv) {
   R v = new R ();
   v.F1 (10);
   v.F2 (15);
  API_Package.Print (new AdaString (“Hello”), v);
}
```

```
}
```

That's it – this works just as if you had written a call directly from Java to Ada!

Compiling and running the code

The last step is to compile and run the code. Since we've already created the GNAT project file for the Ada code, compiling it only requires running gprbuild on it:

```
gprbuild –p –P ada/my_lib.gpr
```

Now you need to adapt either the PATH (on Windows) or LD_LIBRARY_PATH (on Linux/Solaris) in order to list the ada/lib directory, for example:

```
LD_LIBRARY_PATH=`pwd`/ada/lib/:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

The generated java code must be in your CLASSPATH, as well as the main subprogram that you've written, and the lib/ajis.jar file of your AJIS installation. Let's assume that you're running on Linux, and your Java main is at the same location as the api.ads file. You'll need:

```
CLASSPATH=`pwd`:`pwd`/java:<your AJIS
installation>/lib/ajis.jar:$CLASSPATH
```

Compiling and running the Java code is now straightforward:

```
javac My_Main.java
java My_Main
```

Note that the library that we've compiled will be automatically loaded by the generated bound code.

Upcoming Gems will show more advanced usage of GNAT-AJIS, including using callbacks, OOP, exceptions, and memory management.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #56: Creating Ada to Java calls using GNAT-AJIS

Author: Quentin Ochem, AdaCore

Abstract: In a previous Gem we introduced ada2java, which maps an Ada specification to a Java specification, to support calls from Java to Ada. Even though ada2java doesn't support creating Ada bindings of Java specs, it's still possible to use it to support calls from Ada to Java. In this Gem we'll look at a first possibility of doing this, using callbacks (in Ada terms, access-to-subprogram calls).

Let's get started…

Consider the following API:

```
package API is

   type A_Callback is access procedure (Str : String; Val : Integer);

   procedure Process (Str : String; C : A_Callback);

end API;

package body API is
   procedure Process (Str : String; C : A_Callback) is
   begin
      C.all (Str, 1);
      C.all (Str, 2);
   end Process;
end API;
```

In this example, A_Callback is called twice from Process. Our goal here will be to have these calls invoke Java code.

Generating the binding code

As in the previous Gem, generating the binding code is done using the ada2java tool:

```
ada2java api.ads –b test –o ada –c java –L my_lib
```

Details on the meaning of the command arguments can be found in the earlier Gem, or in the ada2java documentation.

Writing the Java code

Looking at the generated code, you'll see that the function Process has been bound as follows:

```
public final class API_Package {
   […]
```

```
   static public void Process (test.Standard.AdaString Str,
test.API.A_Callback C){
   […]
}
```

with test.API.A_Callback being:

```
public abstract class A_Callback {
   […]
   abstract public void A_Callback_Body (test.Standard.AdaString Str,
int Val);
   […]
}
```

Now, all that's needed to implement the Ada access-to-subprogram in Java, is to derive from A_Callback to create a concrete class, implement its A_Callback_Body member, and then give an instance of that object to the Process function.

For example:

```
import test.API.A_Callback;
import test.API.API_Package;
import test.Standard.AdaString;

public class My_Main {
   static class My_Callback extends A_Callback {
      public void A_Callback_Body (AdaString Str, int Val) {
         System.out.println ("Call " + Val + " of " + Str);
      }
   }
   public static void main (String [] argv) {
      API_Package.Process (new AdaString ("Hello"), new My_Callback
());
   }
}
```

And just that easily, you've established a callback, with Java calling Ada and then calling back to Java!

Compiling and running

As in the earlier gem, we've generated the build project file with the –L switch. So following these steps, on Linux, you need to do:

```
gprbuild –p –P ada/my_lib.gpr
LD_LIBRARY_PATH=`pwd`/ada/lib/:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
CLASSPATH=`pwd`:`pwd`/java:<your AJIS
installation>/lib/ajis.jar:$CLASSPATH
javac My_Main.java
java My_Main
```

Now, if you try to run the program, it doesn't work quite as expected. It turns out that an exception is raised at the point of the call to Process, on the Java side. Let's have a look at that in more detail.

Fixing the escapement problem

The problem that we have is that it's not possible to store objects of class A_Callback that have been created from Java – they are not actual library-level access-to-subprogram values, but rather are Java objects. You may want to have a look at the generated Ada glue code if you need more details on this, but the important point is that such objects cannot be used by the native code after the call to Process, so they should not be stored. We'll say that they cannot be "escaped".

Unfortunately, we have no way to ensure that the objects are indeed not escaped, so the AJIS run-time just refuses to pass such objects, in order to prevent the developer from escaping them by mistake. In this very case, we just make two calls. No object will get stored, and calls to Process are perfectly safe. We can give this information to the binding generator by using one of the AJIS annotations:

```
with AJIS.Annotations;

package API is
   type A_Callback is access procedure (Str : String; Val : Integer);

   procedure Process (Str : String; C : A_Callback);
   pragma Annotate (AJIS, Assume_Escaped, False, Process, "C");
```

The pragma indicates that the C parameter of Process should not be assumed as potentially escaped – in other words, the Ada developer takes the responsibility of not allowing it to escape. The AJIS run-time will then allow it to pass non-escapable objects as parameters.

Note that we now have a reference to the AJIS run-time in our working sources. The easiest way to reference that run-time is to use a project file to manage the sources, for example by using the following project file:

```
with "ajis";
project API is
end API;
```

and then the binding can be regenerated:

```
ada2java api.ads –b test –o ada –c java –L my_lib –P api.gpr
```

After recompiling the Ada and Java sources, we relaunch the Java application and calls from Ada to Java now work fine!

In the next Gem we'll look at how to do cross-language dispatching by extending an Ada tagged type in Java.

Related Source Code

Gem #57: Ada / Java cross dispatching

Author: Quentin Ochem, AdaCore

Abstract: In the preceding Ada/Java interfacing Gem, we saw how to create calls from Ada to Java using ada2java and callbacks. We're now going to go one step further, and discuss a cross-language dispatching mechanism that supports extension of an Ada tagged type in Java, allowing the possibility of making dispatching calls equally well from either side.

Let's get started…

Consider the following API:

```
package API is
   type A_Tagged_Type is tagged null record;

   procedure Print_Me (V : A_Tagged_Type; Me : String);

   type An_Ada_Child is new A_Tagged_Type with null record;

   procedure Print_Me (V : An_Ada_Child; Me : String);

   procedure Call_Print_Me (Str : String; Val : A_Tagged_Type'Class);
end API;

package body API is
   procedure Print_Me (V : A_Tagged_Type; Me : String) is
   begin
      Put_Line ("FROM A TAGGED TYPE: " & Me);
   end Print_Me;

   procedure Print_Me (V : An_Ada_Child; Me : String) is
   begin
      Put_Line ("FROM AN ADA CHILD: " & Me);
   end Print_Me;

   procedure Call_Print_Me (Str : String; Val : A_Tagged_Type'Class) is
   begin
      Print_Me (Val, Str);
   end Call_Print_Me;
end API;
```

Note that the call in the Ada Call_Print_Me is dispatching, so if I write in Ada:

```
V1 : A_Tagged_Type;
V2 : An_Ada_Child;
Call_Print_Me ("V1", V1);
Call_Print_Me ("V2", V2);
```

It will call the Print_Me of the actual object, so that the output will be:

```
FROM A TAGGED TYPE: V1
FROM AN ADA CHILD: V2
```

Generating the binding code

As usual, generating the whole binding is done through a simple command:

```
ada2java api.ads –b test –o ada –c java –L my_lib
```

Writing the Java code

We now have two classes, test.API.A_Tagged_Type and test.API.An_Ada_Child, both of them regular non-final java classes, with An_Ada_Child derived explicitly from A_Tagged_Type. So we can write an example similar to what we did in Ada:

```
import test.API_Package;
import test.API.A_Tagged_Type;
import test.API.An_Ada_Child;
import test.Standard.AdaString;

public class My_Main {

public static void main (String [] argv) {
   A_Tagged_Type v1 = new A_Tagged_Type ();
   An_Ada_Child v2 = new An_Ada_Child ();
   API_Package.Call_Print_Me (new AdaString ("V1"), v1);
   API_Package.Call_Print_Me (new AdaString ("V2"), v2);
}
```

which will have exactly the same effect. But we can go even further; it is possible to derive the class A_Tagged_Type in Java, and override its primitive. Let’s do that in a class nested in My_Main, such as:

```
static class A_Java_Child extends A_Tagged_Type {
   public void Print_Me (AdaString Me) {
      System.out.println ("FROM A JAVA CHILD" + Me);
   }
}
```

Note that here, what was an Ada dispatching primitive is now a Java member function. The first parameter of the Ada subprogram, the controlling operand “V”, has been bound to the Java implicit “this” parameter, so that the member ends up having only one explicit parameter instead of two. Incidentally, this kind of construction would not have been possible if the first parameter of the Ada primitive had not been controlling, and ada2java would have warned us about this.

Now we can use instances of this object just as instances of any of its parents, for example:

```
public static void main (String [] argv) {
   A_Tagged_Type v1 = new A_Tagged_Type ();
   An_Ada_Child v2 = new An_Ada_Child ();
   A_Tagged_Type v3 = new A_Java_Child ();
   API_Package.Call_Print_Me (new AdaString ("V1"), v1);
   API_Package.Call_Print_Me (new AdaString ("V2"), v2);
   API_Package.Call_Print_Me (new AdaString ("V3"), v3);
}
```

And just like that, we have put in place a cross-language-dispatching call between Ada and Java. On the Ada side, the code generated by the binding generator and the AJIS run-time will be able to detect that this last object has a type extended in Java, and that the dispatching call has to be resolved using the Java object.

Compiling and running

As in the earlier Ada/Java interfacing Gems, the compile and run commands on Linux breaks down as follows:

```
gprbuild –p –P ada/my_lib.gpr
LD_LIBRARY_PATH=`pwd`/ada/lib/:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
CLASSPATH=`pwd`:`pwd`/java:<your AJIS
installation>/lib/ajis.jar:$CLASSPATH
javac My_Main.java
java My_Main
```

which displays on the screen:

```
FROM A TAGGED TYPE: V1
FROM AN ADA CHILD: V2
FROM A JAVA CHILD: V3
```

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #58: Ada / Java exception handling

Author: Quentin Ochem, AdaCore

Abstract: Ada and Java are two languages that rely heavily on exceptions. A large part of the Ada data model is based on the fact that data is checked at run time, and will raise various kinds of exceptions such as Constraint_Error when constraints are violated. Similarly, there are many cases where Java performs checks that can raise exceptions, among the most common being checks on casts and null dereferences. It is therefore extremely important to support exceptions that are properly propagated from one language to the other and even potentially caught/handled, without having to worry about the language of origin.

Let's get started…

Let's consider the following API:

```
with AJIS.Annotations; use AJIS.Annotations;

package API is

   function Compute (I : Integer) return Integer;

   type An_Operation is access function (I : Integer) return Integer;

   function Indirect_Compute (Data : Integer; Operation : An_Operation)
     return Integer;
   pragma Annotate (AJIS, Assume_Escaped, False, Indirect_Compute,
"Operation");

end API;
```

with the following implementation:

```
package body API is

   function Compute (I : Integer) return Integer is
      Tmp : Natural := I;
   begin
      return Tmp + 1;
   end Compute;

   procedure Indirect_Compute (Data : Integer; Operation :
An_Operation) is
   begin
      return Operation.all (Data);
   exception
      when Constraint_Error =>
         return 0;
   end Indirect_Compute;

end API;
```

The Compute subprogram uses a temporary variable of subtype Natural, and will raise a Constraint_Error if a negative value is passed for I. On the other hand, the Indirect_Compute subprogram catches all Constraint_Errors, and returns 0 if one occurs.

Generating the binding code

As with the previous Ada/Java interfacing Gems, the generation of the binding is straightforward:

```
ada2java api.ads –b test –o ada –c java –L my_lib –P api.gpr
```

Note that you need an api.gpr project file, defined as follows:

```
with "ajis";

project API is

end API;
```

Writing the Java code

Let's now play a bit with this code. As you can see, Java classes are generated for the Ada exceptions, such as test.Standard.Constraint_Error. These exceptions will end up as regular Java exceptions, which we can catch, for example, in the following code:

```
import test.API.API_Package;
import test.Standard.Constraint_Error;

public class My_Main {

   public static void main (String [] argv) {

      try {
         int x = API_Package.Compute (-1);
      } catch (Constraint_Error e) {
      System.out.println (e.getMessage ());
      }
   }
}
```

And just like that, we have an exception raised from Ada, transformed into a Java exception, and finally caught in a Java block. But let's do something a little more ambitious. Let's raise an exception from Ada, have this exception pass through the Java frames, and then end up back in Ada:

```
import test.API.API_Package;
import test.API.An_Operation;
import test.Standard.Constraint_Error;

public class My_Main {
```

```
   public static void main (String [] argv) {
      int x = API_Package.Indirect_Compute (-1,
                  new An_Operation () {
                        public int An_Operation_Body (int I) {
                             API_Package.Compute (I);
                        }
                  }
               );
   }
}
```

So, here we have a callback called from Ada, implemented in Java, that calls the Ada Compute procedure. Let's see what happens in detail. First, Indirect_Compute is called by Java, goes into Ada, which then calls the subprogram passed as an access parameter, resulting in a call to Java. This Java function then calls the Ada Compute operation, which will raise an exception due to passing -1. So at the point of the exception, the call stack looks like this:

```
[1] Java : main
[2] Ada : Indirect_Compute
[3] Java : An_Operation_Body
[4] Ada : Compute <- the exception is raised here
```

The exception first goes from [4] to [3], as in the previous example. In this case however, it is not caught, so it is propagated to the caller, [2], which happens to be Ada code again. Fortunately, the glue code generated between [2] and [3] is able to translate the exception back from Java to the original Ada exception, and continue the propagation. In this case, there's an exception handler in [2], which handles Constraint_Error, and will catch the one initially raised at [4].

Compiling and running

Just as in the other Ada/Java interfacing Gems, the compile and run commands on Linux break down as follows:

```
gprbuild –p –P ada/my_lib.gpr
LD_LIBRARY_PATH=`pwd`/ada/lib/:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
CLASSPATH=`pwd`:`pwd`/java:<your AJIS
installation>/lib/ajis.jar:$CLASSPATH
javac My_Main.java
java My_Main
```

Related Source Code

Gem #59: Generating Ada bindings for C headers

Author: Arnaud Charlet, AdaCore

Abstract: One of the delicate areas of Ada that is often unfamiliar to developers is how to intermix Ada code with other programming languages. While Ada provides very powerful and complete capabilities to interface with other compiled languages such as C, C++, and Fortran, writing the Ada glue code that enables a programmer to use complex and large APIs can be tedious and error-prone.
In this Gem, we will explore a new tool provided by AdaCore to automate the interface generation of C header files through the compiler.

Let's get started…

GNAT now comes with a new experimental binding generator for C and C++ headers which is intended to do 95% of the tedious work of generating Ada specs from C or C++ header files. Note that this is still a work in progress, not designed to generate 100% correct Ada specs.

The code generated uses the Ada 2005 syntax, making it easier to interface with other languages than did previous versions of Ada, in particular via the generation of limited with clauses and anonymous access types.

In this Gem, we will focus on the generation of C bindings. Bindings to C++ will be addressed in a future Gem.

Running the binding generator

The binding generator is part of the GCC compiler that comes with recent versions of GNAT and can be invoked via the -fdump-ada-spec switch, which generates Ada spec files for the header files specified on the command line, and all header files needed by these files transitivitely. For example:

```
$ g++ -c -fdump-ada-spec -C /usr/include/time.h
$ gcc -c -gnat05 *.ads
```

generates, under GNU/Linux, the following files: time_h.ads, bits_time_h.ads, stddef_h.ads, bits_types_h.ads, which correspond to the files /usr/include/time.h, /usr/include/bits/time.h, etc…, and then compiles these Ada specs in Ada 2005 mode.

The -C switch tells GCC to extract comments from headers and attempt to generate corresponding Ada comments.

If you want to generate a single Ada file and not the transitive closure, you can use the -fdump-ada-spec-slim switch instead.

Note that we recommend, where possible, to use the G++ driver to generate bindings, even for most C headers, since this will in general generate better Ada specs. If G++ doesn't work on your C headers because of incompatibilities between C and C++, then you can fall back on using GCC instead.

For an example of better bindings generated from the C++ front end, the name of the parameters (when available) are actually ignored by the C front end. Consider the following C header:

```
extern void foo (int variable);
```

With the C front end, "variable" is ignored, and the above is handled as:

```
extern void foo (int);
```

generating the following subprogram specification:

```
procedure foo (param1 : int);
```

With the C++ front end, the name is available, and we generate:

```
procedure foo (variable : int);
```

In some cases, the generated bindings will be more complete or more meaningful when defining macros, which you can do via the -D switch. This is for example the case with Xlib.h under GNU/Linux:

```
g++ -c -fdump-ada-spec -DXLIB_ILLEGAL_ACCESS -C /usr/include/X11/Xlib.h
```

The above will generate more complete bindings than a straight call without the -DXLIB_ILLEGAL_ACCESS switch.

In other cases, it is not possible to parse a header file in a stand-alone manner, because other include files need to be included first. In this case, the solution is to create a small header file including the needed #include and possible #define directives. For example, to generate Ada bindings for readline/readline.h, you first need to include stdio.h, so you can create a file with the following two lines in, for example, readline1.h:

```
#include <stdio.h>
#include <readline/readline.h>
```

and then generate Ada bindings from this file:

```
$ g++ -c -fdump-ada-spec readline1.h
```

In a future Gem we will talk about generating similar Ada bindings for C++ headers.

Gem #60: Generating Ada bindings for C++ headers

Author: Arnaud Charlet, AdaCore

Abstract: In Gem #59 we saw how simple it is to automatically generate Ada bindings for C header files. In this Gem, we will see that, similarly, it's now also possible to generate Ada bindings for C++ header files.

Let's get started…

Generating bindings for C++ headers is done using the same options as for C headers, again using the G++ driver. This is because the binding generation tool takes advantage of the full C and C++ front ends readily available with GCC, and then translates its internal representation into corresponding Ada code.

In this mode, C++ classes will be mapped to Ada tagged types, constructors will be mapped using the CPP_Constructor pragma, and, when possible, multiple inheritance of abstract classes will be mapped to Ada interfaces.

A complete example

For example, given the following C++ header file called animals.h:

```
class Carnivore {
public:
  virtual int Number_Of_Teeth () = 0;
};

class Domestic {
public:
  virtual void Set_Owner (char* Name) = 0;
};

class Animal {
public:
  int Age_Count;
  virtual void Set_Age (int New_Age);
};

class Dog : Animal, Carnivore, Domestic {
public:
  int Tooth_Count;
  char *Owner;

  virtual int  Number_Of_Teeth ();
  virtual void Set_Owner (char* Name);

  Dog();
};
```

We generate an Ada package with:

```
$ g++ -c -fdump-ada-spec animals.h
```

which generates a file animals_h.ads that has the following contents:

```
package animals_h is

   package Class_Carnivore is
     type Carnivore is limited interface;
     pragma Import (CPP, Carnivore);

     function Number_Of_Teeth (this : access Carnivore) return int is
abstract;
   end;
   use Class_Carnivore;

   package Class_Domestic is
     type Domestic is limited interface;
     pragma Import (CPP, Domestic);

     procedure Set_Owner
       (this : access Domestic;
        Name : Interfaces.C.Strings.chars_ptr) is abstract;
   end;
   use Class_Domestic;

   package Class_Animal is
     type Animal is tagged limited record
       Age_Count : aliased int;
     end record;
     pragma Import (CPP, Animal);

     procedure Set_Age (this : access Animal; New_Age : int);
     pragma Import (CPP, Set_Age, "_ZN6Animal7Set_AgeEi");
   end;
   use Class_Animal;

   package Class_Dog is
     type Dog is new Animal and Carnivore and Domestic with record
       Tooth_Count : aliased int;
       Owner : Interfaces.C.Strings.chars_ptr;
     end record;
     pragma Import (CPP, Dog);

     function Number_Of_Teeth (this : access Dog) return int;
     pragma Import (CPP, Number_Of_Teeth,
"_ZN3Dog15Number_Of_TeethEv");

     procedure Set_Owner
       (this : access Dog; Name : Interfaces.C.Strings.chars_ptr);
     pragma Import (CPP, Set_Owner, "_ZN3Dog9Set_OwnerEPc");

     function New_Dog return Dog'Class;
     pragma CPP_Constructor (New_Dog, "_ZN3DogC1Ev");
   end;
   use Class_Dog;
```

```
end animals_h;
```

As you can see, C++ classes are mapped to Ada tagged types and even interfaces when possible. Another advantage of directly using the C++ compiler to generate bindings is that the information about C++ mangling is readily available and can be generated easily, while writing it manually can be tedious and error prone.

The C++ mangling is the low-level name chosen by the C++ compiler for a given function, such as "_ZN3Dog9Set_OwnerEPc" in the case of procedure Class_Dog.Set_Owner in the example above.

In the next Gem, we will explore in more detail how the CPP_Constructor pragma works and how to combine it with Ada features.

Related Source Code

Gem #61: Interfacing with C++ constructors

Author: Javier Miranda, Arnaud Charlet, AdaCore

Abstract: In the previous Gem about generating bindings from C++ headers, we mentioned, without going into details, how to interface with C++ constructors in Ada using the CPP_Constructor pragma.

In this Gem we present some common uses of C++ constructors in mixed-language programs in GNAT, and in the next Gem, we will show the use of some powerful Ada 2005 features in conjunction with C++ constructors.

Let's get started…

Let's assume that we need to interface with the following C++ class:

```
class Root {
public:
   int  a_value;
   int  b_value;
   virtual int Get_Value ();
   Root();              // Default constructor
   Root(int v);         // 1st non-default constructor
   Root(int v, int w);  // 2nd non-default constructor
};
```

Using the automatic binding generator, or writing manually, we can obtain the corresponding package spec:

```
with Interfaces.C; use Interfaces.C;
package Pkg_Root is

   type Root is tagged limited record
       A_Value : int;
       B_Value : int;
   end record;
   pragma Import (CPP, Root);

   function Get_Value (Obj : Root) return int;
   pragma Import (CPP, Get_Value);

   function New_Root return Root'Class;
   pragma Cpp_Constructor (New_Root, "_ZN4RootC1Ev");

   function New_Root (v : int) return Root'Class;
   pragma Cpp_Constructor (New_Root, "_ZN4RootC1Ei");

   function New_Root (v, w : int) return Root'Class;
   pragma Cpp_Constructor (New_Root, "_ZN4RootC1Eii");

end Pkg_Root;
```

On the Ada side, the constructor is represented by a function (whose name is arbitrary) that returns the class-wide type corresponding to the imported C++ class. The return type is required to be class-wide rather than the specific Root type so that the function will not be treated as a primitive dispatching operation of the type. Although the constructor is described as a function, it is typically a procedure with an extra implicit argument (the object being initialized) at the implementation level. GNAT issues the appropriate call, whatever it is, to initialize the object.

Constructors can appear in the following contexts:

- As the initialization expression of an object of type T
- As the initialization expression of a record component of type T
- In a limited aggregate
- In a limited aggregate used as the expression of a return statement

Note that all of the above contexts are places where Ada 2005 allows limited aggregates and calls to functions with limited results to appear, and in fact it's necessary to enable the Ada 2005 compilation mode (-gnat05) to make use of this feature.

In a declaration of an object whose type is a class imported from C++, either the default C++ constructor is implicitly called by GNAT, or else the required C++ constructor must be explicitly called in the expression that initializes the object. For example:

```
Obj1 : Root;
Obj2 : Root := New_Root;
Obj3 : Root := New_Root (v => 10);
Obj4 : Root := New_Root (30, 40);
```

The first two declarations are equivalent: in both cases, the default C++ constructor is invoked (in the former case the call to the constructor is implicit, and in the latter case the call is explicit in the object declaration). Obj3 is initialized by the C++ nondefault constructor that takes an integer argument, and Obj4 is initialized by the nondefault C++ constructor that takes two integers.

It's worth pointing out that normally it's not permitted to call a class-wide function to initialize an object of a specific type, and it's only in the case of these special imported constructor functions that the compiler allows this usage.

In the next Gem we will explore how to derive and extend imported C++ classes on the Ada side, and show uses of constructors in Ada 2005 extended return and limited aggregates.

Related Source Code

Gem #62: C++ constructors and Ada 2005

Author: Javier Miranda, Arnaud Charlet, AdaCore

Abstract: In the previous Gem, we explored how to interface and make simple use of C++ constructors on the Ada side.

In this Gem, we detail more advanced constructs related to Ada 2005 and C++ constructors.

Let's get started…

Continuing from the example discussed in the last Gem, let's derive the imported C++ class on the Ada side. For example:

```
type DT is limited new Root with record
   C_Value : Natural := 2009;
end record;
```

In this case, the components DT inherited from the C++ side must be initialized by a C++ constructor, and the additional Ada components of type DT are initialized by GNAT. The initialization of such an object is done either by default, or by means of a function returning an aggregate of type DT, or by means of an extension aggregate:

```
Obj5 : DT;
Obj6 : DT := Function_Returning_DT (50);
Obj7 : DT := (New_Root (30, 40) with C_Value => 50);
```

The declaration of Obj5 invokes the default constructors: the C++ default constructor of the parent type takes care of the initialization of the components inherited from Root, and GNAT takes care of the default initialization of the additional Ada components of type DT (that is, C_Value is initialized to value 2009). The order of invocation of the constructors is consistent with the order of elaboration required by Ada and C++, meaning that the constructor of the parent type is always called before the constructor of the derived type.

Let's now consider a record that has components whose type is imported from C++. For example:

```
type Rec1 is limited record
   Data1 : Root := New_Root (10);
   Value : Natural := 1000;
end record;

type Rec2 (D : Integer := 20) is limited record
   Rec   : Rec1;
   Data2 : Root := New_Root (D, 30);
end record;
```

The initialization of an object of type Rec2 will call the nondefault C++ constructors specified for the imported components. For example:

```
Obj8 : Rec2 (40);
```

Using Ada 2005 we can use limited aggregates to initialize an object invoking C++ constructors that differ from those specified in the type declarations. For example:

```
Obj9 : Rec2 := (Rec => (Data1 => New_Root (15, 16),
                        others => <>),
                others => <>);
```

The above declaration uses an Ada 2005 limited aggregate to initialize Obj9, and the C++ constructor that has two integer arguments is invoked to initialize the Data1 component instead of the constructor specified in the declaration of type Rec1. In Ada 2005, the box in the aggregate indicates that unspecified components are initialized using the expression (if any) available in the component declaration. That is, in this case discriminant D is initialized to value 20, Value is initialized to value 1000, and the non-default C++ constructor that handles two integers takes care of initializing component Data2 with values 20 and 30.

In Ada 2005, we can use the extended return statement to build the Ada equivalent of a C++ nondefault constructor. For example:

```
function New_Rec2 (V : Integer) return Rec2 is
begin
   return Obj : Rec2 := (Rec => (Data1  => New_Root (V, 20),
                                 others => <>),
                         others => <>) do
      --  Further actions required for construction of
      --  objects of type Rec2
      …
   end record;
end New_Rec2;
```

In this example, the extended return statement construct is used to build in place the returned object, whose components are initialized by means of a limited aggregate. Any further actions associated with the constructor can be given within the statement part of the extended return.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #63: The Effect of Pragma Suppress

Author: Gary Dismukes, AdaCore

Abstract: The features of Ada have generally been designed to prevent violating the properties of data types, enforced either by compile-time rules or, in the case of dynamic properties, by using run-time checks. Ada allows run-time checks to be suppressed, but not with the intent of allowing programmers to subvert the type system.

Let's get started…

One of Ada's key strengths has always been its strong typing. The language imposes stringent checking of type and subtype properties to help prevent accidental violations of the type system that are a common source of program bugs in other less-strict languages such as C. This is done using a combination of compile-time restrictions (legality rules), that prohibit mixing values of different types, together with run-time checks to catch violations of various dynamic properties. Examples are checking values against subtype constraints and preventing dereferences of null access values.

At the same time, Ada does provide certain "loophole" features, such as Unchecked_Conversion, that allow selective bypassing of the normal safety features, which is sometimes necessary when interfacing with hardware or code written in other languages.

Ada also permits explicit suppression of the run-time checks that are there to ensure that various properties of objects are not violated. This suppression can be done using pragma Suppress, as well as by using a compile-time switch on most implementations (in the case of GNAT, with the -gnatp switch).

In addition to allowing all checks to be suppressed, Pragma Suppress supports suppression of specific forms of check, such as Index_Check for array indexing, Range_Check for scalar bounds checking, and Access_Check for dereferencing of access values. (See section 11.5 of the Ada Reference Manual for further details.)

Here's a simple example of suppressing index checks within a specific subprogram:

```
procedure Main is
   procedure Sort_Array (A : in out Some_Array) is
      pragma Suppress (Index_Check);  — eliminate check overhead
   begin
      …
   end Sort_Array;
end Main;
```

Unlike a feature such as Unchecked_Conversion, however, the purpose of check suppression is not to enable programs to subvert the type system, though many programmers seem to have that misconception.

What's important to understand about pragma Suppress is that it only gives permission to the implementation to remove checks, but doesn't require such elimination. The intention of Suppress is not to allow bypassing of Ada semantics, but rather to improve efficiency, and the Ada Reference Manual has a clear statement to that effect in the note in RM-11.5, paragraph 29:

There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress should be used only for efficiency reasons.

There is associated Implementation Advice that recommends that implementations should minimize the code executed for checks that have been suppressed, but it's still the responsibility of the programmer to ensure that the correct functioning of the program doesn't depend on checks not being performed.

There are various reasons why a compiler might choose not to remove a check. On some hardware, certain checks may be essentially free, such as null pointer checks or arithmetic overflow, and it might be impractical or add extra cost to suppress the check. Another example where it wouldn't make sense to remove checks is for an operation implemented by a call to a run-time routine, where the check might be only a small part of a more expensive operation done out of line.

Furthermore, in many cases GNAT can determine at compile time that a given run-time check is guaranteed to be violated. In such situations, it gives a warning that an exception will be raised, and generates code specifically to raise the exception. Here's an example:

```
X : Integer range 1..10 := …;

..

if A > B then
   X := X + 1;
   ..
end if;
```

For the assignment incrementing X, the compiler will normally generate machine code equivalent to:

```
Temp := X + 1;
if Temp > 10 then
   raise Constraint_Error;
end if;
X := Temp;
```

If range checks are suppressed, then the compiler can just generate the increment and assignment. However, if the compiler is able to somehow prove that X = 10 at this point, it will issue a warning, and replace the entire assignment with simply:

```
raise Constraint_Error;
```

even though checks are suppressed. This is appropriate, because (1) we don't care about the efficiency of buggy code, and (2) there is no "extra" cost to the check, because if we reach that point, the code will unconditionally fail.

One other important thing to note about checks and pragma Suppress is this statement in the Ada RM (RM-11.5, paragraph 26):

If a given check has been suppressed, and the corresponding error situation occurs, the execution of the program is erroneous.

In Ada, erroneous execution is a bad situation to be in, because it means that the execution of your program could have arbitrary nasty effects, such as unintended overwriting of memory. Note also that a program whose "correct" execution somehow depends on a given check being suppressed might work as the programmer expects, but could still fail when compiled with a different compiler, or for a different target, or even with a newer version of the same compiler. Other changes such as switching on optimization or making a change to a totally unrelated part of the code could also cause the code to start failing.

So it's definitely not wise to write code that relies on checks being removed. In fact, it really only makes sense to suppress checks once there's good reason to believe that the checks can't fail, as a result of testing or other analysis. Otherwise, you're removing an important safety feature of Ada that's intended to help catch bugs.

Related Source Code

Gem #64: Handling Multiple-Unit Source Files

Author: Emmanuel Briot, AdaCore

Abstract: This Gem describes how to compile applications in GNAT when source files contain multiple units. The preferred approach is to split source files, and here we describes how this can be done, although GNAT also provides a workaround that allows you to keep your existing files.

Let's get started…

In lots of legacy systems that were initially developed with technologies other than GNAT, a single source file might include several Ada units (or both spec and body of a single unit).

Where possible, it is best to follow GNAT's traditional approach and split these files into several files. The main advantage this provides is to limit the amount of recompilation that gnatmake and gprbuild will do. When several units are found in the same file, modifying any of them will force recompilation of all files that depend on any of the units found in the file. In particular, changing the body of a unit when the spec is in the same source file will force recompilation of all units that depend on it. This is contrary to the whole concept of separate specifications.

This is also the recommended method in the Ada Quality and Style manual, so this should really be the preferred approach where possible.

Such splitting of source files can be done through the gnatchop command line tool. If you want to keep the original (multiple-unit) source files and keep modifying these, we recommend using the "-r" switch, which will force GNAT's error messages to reference the original (unsplit) file.

If however you do not intend to keep them, you should not use "-r".

Most likely the "-w" switch will also come in handy, to force overwriting of previously split files. Also, since you are recreating the source files that gnatmake sees, you should use the "-m" switch for gnatmake so that it doesn't pay attention to timestamps. Otherwise you would end up recompiling all files all the time.

Calling gnatchop cannot be done automatically by gnatmake. Instead, you should have a script or a Makefile that first calls gnatchop, and then spawns gnatmake to do the actual compilation.

However, in some cases it isn't possible to split the source files. Most often, this is because the files are under configuration control and you want to preserve history. In other cases, this is because you still want to compile with your older compiler.

GNAT provides a workaround in cases where you cannot split your source files and cannot or do not want to change your build methods to use gnatchop. Let us emphasize that this is really considered as only a workaround, and is not fully supported by all the tools. For instance, the “-gnatD” and “-gnatR” switches are currently not compatible with the pragmas described below. More important perhaps, GPS will have some known limitations when using multiple-unit source files (cross-references will not work in a lot of cases, and compiling the current file will fail if it contains multiple units)

You have to let GNAT know about your source-file organization, which can be done in several ways.

Method 1: If you are using project files, you can add or create a package Naming inside your project file, and use a syntax like:

```
package Naming is
  for Specification ("unit1") use "file1.ada" at 1;
  for Implementation ("unit1") use "file1.ada" at 2;
end Naming;
```

The index after the **at** starts at 1 and indicates the position of the unit inside the source file.

You can either create this Naming package manually by editing the project file, or else by using the gnatname command-line tool. This tool will traverse all the directories you pass it via its “-d” switch, check each file in them, and determine what unit or units they contain. It will then create a new project file called my_project_naming.gpr, and modify the project you specified on the command line in a manner similar to:

```
with "my_project_naming";
project My_Project is
   package Naming renames My_Project_Naming.Naming;
end My_Project;
```

GPS was recently modified to properly support such project files, and properly preserve the existing “**at**” information when you edit a project, as well as to allow you to create new naming exceptions directly from the graphical interface.

Method 2: If you are not using project files yet, you can specify the same information by using pragmas in a configuration file. Through its “-c” switch, gnatname is also able to generate this file of pragmas. The general form of the configuration file is:

```
pragma Source_File_Name
  (Unit1, Spec_File_Name => "file1.ada", Index => 1);
pragma Source_File_Name
  (Unit1, Body_File_Name => "file1.ada", Index => 2);
```

where the information is similar to what was specified in the project file example above.

There is a difference between using these project attributes or pragmas, and using “gnatchop -r” as we described in the first part: while there is a single command to spawn, and no duplication of source files with the pragmas, you still have the issue that gnatmake will recompile everything that depends on any unit in the file. When using gnatchop, however, that problem does not exist, because gnatmake sees single-unit source files. Also, the pragma-based method is not fully supported by all the tools yet, so the preferred approach is really to split your files (and preferably replace the multiple-unit source files with the resulting single-unit files).

Related Source Code